MATH ASSIGNMENT 2

Q1) Gram-Schmidt Algorithm and QR decomposition (5 marks)

i) Write a code to generate a random matrix A of size m × n with m > n and calculate its Frobenius norm, $\|\cdot\|_F$. The entries of A must be of the form r.dddd (example 5.4316). The inputs are the positive integers m and n and the output should display the the dimensions and the calculated norm value. Deliverable(s) : The code with the desired input and output (0.5)

Used Libraries

In [ ]:
```
import random
import math
import copy
from math import sqrt
from numpy import array, zeros
```

In [ ]:
```
# Function to return the Frobenius, Norm of the given matrix
# Generate Matrix
def generate_matrix():
    nRows = int(input("NUMBER OF ROWS = "))
    nCols=int(input("NUMBER OF COLUMNS = "))
    matx = zeros((nRows,nCols))
    for row_elm in range(nRows):
        for col_elm in range(nCols):
            number = round(random.random(),4)
            while True:
                if not len(str(number).split('.')[-1]) == 4:
                    number = round(random.random(),4)
                else:
                    break
            matx[row_elm][col_elm] = number
    return matx
mat=generate_matrix()
origMatrix=mat
print("\nGENERATED MATRIX = \n{}".format(mat))

def frobeniusNorm(mat):
    sumSq = 0
    rows, columns = mat.shape
    for row_elm in range(rows):
        for col_elm in range(columns):
            sumSq += mat[row_elm][col_elm] ** 2
    return math.sqrt(sumSq)
print("\nFROBENIOUS NORM OF GENERATED MATRIX = ",frobeniusNorm(mat))
```

```
GENERATED MATRIX =
[[0.2388 0.5309 0.7278 0.8709 0.3535]
 [0.4696 0.0587 0.6637 0.1318 0.7937]
 [0.2436 0.9215 0.8161 0.7625 0.8184]
 [0.8433 0.6648 0.0934 0.3146 0.9271]
 [0.0773 0.6369 0.2679 0.7405 0.8747]
 [0.3038 0.7564 0.7377 0.5718 0.7079]
 [0.0362 0.4421 0.9934 0.1329 0.3246]]

FROBENIOUS NORM OF GENERATED MATRIX =  3.626294097284444
```

ii) Write a code to decide if Gram-Schmidt Algorithm can be applied to columns of a given matrix A through calculation of rank. The code should print appropriate messages indicating whether Gram-Schmidt is applicable on columns of the matrix or not. Deliverable(s) : The code that performs the test. (1)

In [ ]:

```
class rankMatrix(object):
    def __init__(self, Matrix):
        self.R = len(Matrix)
        self.C = len(Matrix[0])

    def swap(self, Matrix, row1, row2, col):
        for i in range(col):
            temp = Matrix[row1][i]
            Matrix[row1][i] = Matrix[row2][i]
            Matrix[row2][i] = temp

    def Display(self, Matrix, row, col):
        for i in range(row):
            for j in range(col):
                print (" " + str(Matrix[i][j]))
            print ('\n')

    def rankOfMatrix(self, Matrix):
        rank = self.C
        for row in range(0, rank, 1):
            if Matrix[row][row] != 0:
                for col in range(0, self.R, 1):
                    if col != row:
                        multiplier = (Matrix[col][row] / Matrix[row][row])
                        for i in range(rank):
                            Matrix[col][i] -= (multiplier * Matrix[row][i])
            else:
                reduce = True
                for i in range(row + 1, self.R, 1):
                    if Matrix[i][row] != 0:
                        self.swap(Matrix, row, i, rank)
                        reduce = False
                        break
                if reduce:
                    rank -= 1
                    for i in range(0, self.R, 1):
                        Matrix[i][row] = Matrix[i][rank]
                row -= 1
        return rank

def gramSchmidt(mat):
    mat_original = copy.deepcopy(mat)
    rankobj = rankMatrix(mat)
    rank = rankobj.rankOfMatrix(mat)
    columns = mat.shape[1]
    if rank == columns:
        print("\nGRAM SCHMIDT PROCESS IS APPLICABLE. COLUMNS: {}, RANK: {}".format(colu
    else:
        print("\nGRAM SCHMIDT PROCESS IS NOT APPLICABLE. COLUMNS: {}, RANK: {}".format(
print("\nGENERATED MATRIX = \n{}".format(origMatrix))
gramSchmidt(mat)
```

```
GENERATED MATRIX =
[[0.2388 0.5309 0.7278 0.8709 0.3535]
 [0.4696 0.0587 0.6637 0.1318 0.7937]
```

```
[0.2436 0.9215 0.8161 0.7625 0.8184]
[0.8433 0.6648 0.0934 0.3146 0.9271]
[0.0773 0.6369 0.2679 0.7405 0.8747]
[0.3038 0.7564 0.7377 0.5718 0.7079]
[0.0362 0.4421 0.9934 0.1329 0.3246]]

GRAM SCHMIDT PROCESS IS APPLICABLE. COLUMNS: 5, RANK: 5
```

iii) Write a code to generate the orthogonal matrix Q from a matrix A by performing the Gram-Schmidt orthogonalization method. Ensure that A has linearly independent columns by checking the rank. Keep generating A until the linear independence is obtained. Deliverable(s) : The code that produces matrix Q from A (1)

In [ ]:

```
def calculate_l2_norm(sample):
    sum_up = 0
    for i in sample:
        sum_up += i **2
    return math.sqrt(sum_up)

def gram_schmidt_Q_vector(mat):
    rows, cols = mat.shape
    Q_vec = zeros((rows, cols))
    for j in range(cols):
        v = mat[:, j]
        for k in range(j):
            q = Q_vec[:, k]
            v = v - q.dot(v) * q

        Q_vec[:, j] = v / calculate_l2_norm(v)
    return Q_vec
matx=origMatrix
print("\nGENERATED MATRIX = \n{}".format(origMatrix))
print("\nGRAM SCHMIDT ORTHOGONALISATION FOR THE GENERATED MATRIX = \n{}\n".format(gram_
```

```
GENERATED MATRIX =
[[ 2.38800000e-01  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 5.73397307e-17 -9.85314405e-01  0.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 6.41857227e-18  0.00000000e+00 -2.22277429e-01  0.00000000e+00
   0.00000000e+00]
 [ 2.15737858e-16  0.00000000e+00  0.00000000e+00  4.25670902e+00
  -4.44089210e-16]
 [ 5.00790723e-18  0.00000000e+00  0.00000000e+00  0.00000000e+00
   8.00878201e-01]
 [ 2.48375269e-17  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 4.48049857e-17  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00]]

GRAM SCHMIDT ORTHOGONALISATION FOR THE GENERATED MATRIX =
[[ 1.00000000e+00  2.40116125e-16  2.68784433e-17 -9.03424865e-16
  -2.09711358e-17]
 [ 2.40116125e-16 -1.00000000e+00  7.69565053e-49  4.11497021e-47
  -1.70869369e-48]
 [ 2.68784433e-17  6.45394766e-33 -1.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 9.03424865e-16  2.16926878e-31  2.42826540e-32  1.00000000e+00
   0.00000000e+00]
 [ 2.09711358e-17  5.03550787e-33  5.63671485e-34 -1.89458455e-32
```

```
  1.00000000e+00]
[ 1.04009744e-16  2.49744167e-32  2.79562001e-33 -9.39649890e-32
 -2.18120247e-33]
[ 1.87625568e-16  4.50519244e-32  5.04308320e-33 -1.69505604e-31
 -3.93472127e-33]]
```

iv) Write a code to create a QR decomposition of the matrix A by utilizing the code developed in the previous sub-parts of this question. Find the matrices Q and R and then display the value ‖A − (Q.R)‖F , where ‖ · ‖F is the Frobenius norm. The code should also display the total number of additions, multiplications and divisions to find the result. Deliverable(s) : The code with the said input and output. The results obtained for A generated with m = 7 and n = 5 with random entries described above. (2.5)

In [ ]:

```
def frobeniusNorm(mat):
    sumSq = 0
    rows, columns = mat.shape
    for row_elm in range(rows):
        for col_elm in range(columns):
            sumSq += mat[row_elm][col_elm] ** 2
    return math.sqrt(sumSq)

def calculate_l2_norm(sample):
    sum_up = 0
    for i in sample:
        sum_up += i **2
    return math.sqrt(sum_up)

def gram_schmidt_Q_vector(mat):
    rows, cols = mat.shape
    Q_vec = zeros((rows, cols))
    for j in range(cols):
        v = mat[:, j]
        for k in range(j):
            q = Q_vec[:, k]
            v = v - q.dot(v) * q

        Q_vec[:, j] = v / calculate_l2_norm(v)
    return Q_vec

def compute_R_vector(matrix, Q_Vector):
    return Q_Vector.T @ matrix


def total_operation(matrix):
    m,n = matrix.shape
    addition_Q = ((n * (m-1) * m)/2) + (n-1)*m
    division_Q = m*n
    multiplication_Q = n*m**2
    total_operation_Q = addition_Q + division_Q + multiplication_Q

    addition_R = (m*(m+1)*(n-1))/2
    multiplcation_R = (m*(m+1)*n)/2
    total_operation_R = addition_R + multiplcation_R

    total_operation = total_operation_Q + total_operation_R
    print("TOTAL NUMBER OF ADDITION, DIVISION, MULTIPLICATION ARE = {}".format(int(tota

A=origMatrix
print("\nGENERATED MATRIX = \n{}".format(origMatrix))
```

```
Q = gram_schmidt_Q_vector(A)
R = compute_R_vector(A, Q)
print("\nVALUE OF ||A - (Q.R)|| (FROBENIOUS) = {}\n".format(frobeniusNorm(A - Q.dot(R))
total_operation(A)
```

```
GENERATED MATRIX =
[[ 2.38800000e-01  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 5.73397307e-17 -9.85314405e-01  0.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 6.41857227e-18  0.00000000e+00 -2.22277429e-01  0.00000000e+00
   0.00000000e+00]
 [ 2.15737858e-16  0.00000000e+00  0.00000000e+00  4.25670902e+00
  -4.44089210e-16]
 [ 5.00790723e-18  0.00000000e+00  0.00000000e+00  0.00000000e+00
   8.00878201e-01]
 [ 2.48375269e-17  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00]
 [ 4.48049857e-17  0.00000000e+00  0.00000000e+00  0.00000000e+00
   0.00000000e+00]]

VALUE OF ||A - (Q.R)|| (FROBENIOUS) = 2.6429268858037723e-31

TOTAL NUMBER OF ADDITION, DIVISION, MULTIPLICATION ARE = 665
```

Q2) Gradient Descent Algorithm (2 marks)

i) Consider the last 4 digits of your mobile number (Note : In case there is a 0 in one of the digits replace it by 3). Let it be n1n2n3n4. Generate a random matrix A of size n1n2 × n3n4. For example, if the last four digits are 2311, generate a random matrix of size 23 × 11. Write a code to calculate the l∞ norm of this matrix. Deliverable(s) : The code that generates the results. (0.5)

In [ ]:

```
import numpy as np

def generate_matrix(number):
    rows, columns = int(number[-4:-2]), int(number[-2:])
    return np.random.rand(rows, columns)

def calculate_l_infinity(matrix):
    return np.linalg.norm(matrix, ord =np.inf, axis=1)

sample_number = "9998889911"
A = generate_matrix(sample_number)
print("L INFINITY NORM FOR THE FOLLOWING IS = \n{}\n".format(calculate_l_infinity(A)))
```

```
L INFINITY NORM FOR THE FOLLOWING IS =
[0.99506383 0.77817228 0.83949235 0.99364432 0.86287342 0.92361626
 0.99402894 0.85511678 0.96846343 0.66001674 0.99855435 0.98153173
 0.98952778 0.9840933  0.98579246 0.93105175 0.92043711 0.88978195
 0.99363202 0.96284167 0.92188141 0.99852021 0.96177076 0.86351214
 0.97589262 0.95347471 0.90493373 0.86430495 0.97650893 0.81849002
 0.95099141 0.76837257 0.88561653 0.85316943 0.9361576  0.9517452
 0.65411192 0.76596607 0.78049774 0.95053994 0.90461122 0.98261539
 0.99025697 0.95779585 0.92997269 0.97045854 0.91883631 0.92735381
 0.94801072 0.94520446 0.80045531 0.70513486 0.96698242 0.99422564
 0.97477463 0.92006339 0.99459765 0.82269199 0.71982099 0.88280192
 0.99156669 0.8508252  0.92053754 0.97665354 0.86976371 0.84523225
 0.9502606  0.86887601 0.67369921 0.92274818 0.78651797 0.78808447
 0.93569337 0.88498125 0.98689479 0.98912545 0.90458095 0.83280646
 0.91335354 0.76131801 0.96267257 0.80146942 0.93550991 0.86130626
```

```
0.96237123 0.95666838 0.95972237 0.99644673 0.95837881 0.89812082
0.98169983 0.90404334 0.97314095 0.8169185  0.94064079 0.87315436
0.95910987 0.85581192 0.75661426]
```

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

A = np.random.randint(10,size=(11,11)) # random matrix A of size n1n2 × n3n4
b = np.random.randint(10,size=(11,1)) # vector b of size n1n2 × 1
print('A\n', A, '\n')
#print('b\n', b, '\n')
n1n2, n3n4 = A.shape
rowsums = []
for i in np.arange(n1n2):
    v = np.sum(np.absolute(A[i, :]))
    rowsums.append(v)
print("L∞ NORM OF MATRIX A OF SIZE N1N2 × N3N4 :",np.max(rowsums))
```

```
A
 [[1 0 5 2 2 3 8 6 0 9 6]
 [1 7 6 4 3 6 2 4 6 0 1]
 [6 9 8 5 5 5 3 2 3 3 8]
 [6 6 8 8 2 2 4 6 7 6 2]
 [2 9 4 2 0 3 5 0 8 7 3]
 [8 0 7 1 7 9 7 3 0 0 8]
 [9 2 7 0 5 1 6 2 7 0 8]
 [8 1 7 5 6 2 7 0 2 1 3]
 [6 0 6 4 4 4 8 7 9 0 7]
 [3 4 7 6 3 4 4 4 2 4 2]
 [8 3 1 5 2 7 3 5 8 6 1]]

L∞ NORM OF MATRIX A OF SIZE N1N2 × N3N4 : 57
```

ii) Generate a random vector b of size n1n2 × 1 and consider the function f(x) = 12‖Ax − b‖22 where ‖ · ‖2 is the vector ℓ2 norm. Its gradient is given to be ∇f(x) = A⊤Ax − A⊤b. Write a code to find the local minima of this function by using the gradient descent algorithm (by using the gradient expression given to you). The step size τ in the iteration xk+1 = xk − τ∇f(xk) should be chosen by the formula τ =gTk gk/gTkATAgk where gk = ∇f(xk) = A⊤Axk − A⊤b. The algorithm should execute until ‖xk − xk−1‖2 < 10−4. Deliverable(s) : The code that finds the minimum of the given function and the expression for τ . The values of xk and f(xk) should be stored in a file.

In [ ]:

```
def Gradient_Descent(gradient, start, learn_rate, n_iterations=50, tolerance=1e-5):
    vector = start
    for _ in range(n_iterations):
        diff = -learn_rate * gradient(vector)
        if np.all(np.abs(diff) <= tolerance):
            break
        vector += diff
    return vector

def tau(gk,A):
    numarator = gk.transpose() @ gk
    denominator = gk.transpose() @ A.transpose() @ A @ gk
    tau = np.divide(numarator, denominator) # τ = gk^T gk / gk^T A^T Agk
    return tau

def GDIterations(A,b,error=1e-5):
    x = np.random.randint(10,size=(11,1))
```

```
    norm = np.round(a= np.linalg.norm(x=x, ord=2), decimals=5)
    function = 0.5 * np.square(A @ x - b) # f(x) = 1/2‖Ax − b‖^2 2
    function_norm = np.round(a=np.linalg.norm(x=function, ord=2), decimals=5)
    iteration = 0
    x_list = list()
    fx_list = list()
    while abs(norm) > error:
        df = A.transpose() @ A @ x - A.transpose() @ b # ∇f(x) = ATAx − ATb.
        tau_value = tau(gk=df, A=A)
        xprev = x
        x = x - tau_value * df
        x_minus_prev = x - xprev
        function = 0.5 * np.square(A @ x - b)
        function_norm = np.round(a=np.linalg.norm(x=function, ord=2), decimals=5)
        norm = np.round(a=np.linalg.norm(x=x_minus_prev, ord=2), decimals=5)
        x_list.append(x)
        fx_list.append(function_norm)
        iteration = iteration + 1
    return x_list, fx_list, iteration, tau_value, df

x_list, fx_list, iteration, tau_value,df = GDIterations(A=A,b=b,error=1e-5)
itCount=iteration
fxlst=fx_list
vector=Gradient_Descent(gradient=lambda v: df,start=np.random.randn(11,1), learn_rate=t
print("GENERATED VECTOR = \n",vector)
```

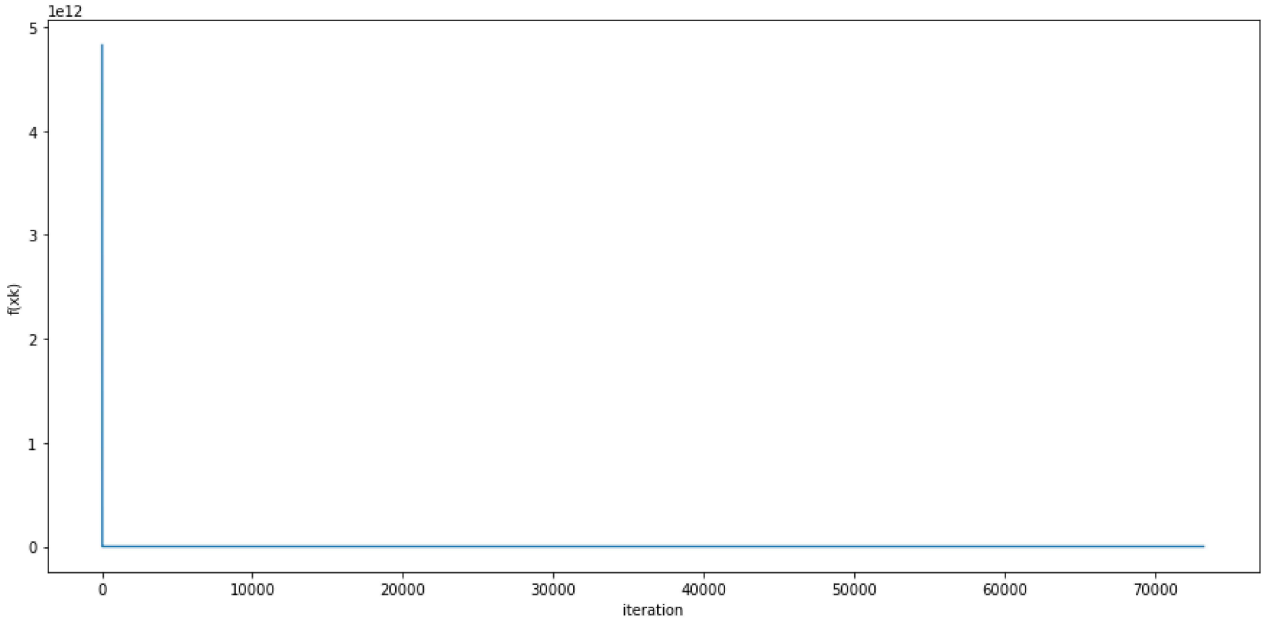
```
GENERATED VECTOR =
 [[-1.00096879]
 [-0.33523148]
 [-0.18356996]
 [ 1.54482035]
 [ 2.13108918]
 [-0.19947632]
 [ 0.9992991 ]
 [-0.66658753]
 [-0.80693726]
 [-0.85148957]
 [ 0.39504876]]
```

iii) Generate the graph of f(xk) vs k where k is the iteration number and xk is the current estimate of x at iteration k. This graph should convey the decreasing nature of function values.
Deliverable(s) : The graph that is generated.

In [ ]:

```
# Generate the graph of f(xk) vs k where k is the iteration number and xk is the curren

step_size = [x for x in range(itCount)]
plt.figure(figsize =[15,7])
sns.lineplot(x = step_size, y = fx_list)
plt.xlabel("iteration")
plt.ylabel("f(xk)")
plt.show()
```


1e12
5
4
3
2
1
0
f(xk)
0
10000
20000
30000
40000
50000
60000
70000
iteration